AN1140: *Bluetooth*® Mesh for Android ADK

This document describes how to get started with Bluetooth mesh application development for Android smart phones and tablets using the Silicon Labs Bluetooth mesh for Android Application Development Kit (ADK).

The document also provides a high level architecture overview of the Silicon Labs Bluetooth mesh library, how it relates to the Bluetooth LE stack provided by the Android operating system and what APIs are available. It also contains code snippets and explanations for the most common Bluetooth mesh use cases.

Warning: The APIs in the Bluetooth mesh for Android ADK are being updated. The updates and documentation will be available in early 2019. Silicon Labs recommends that you do not begin development until those updates have been released.

KEY POINTS

- Introduction to the Silicon Labs' Bluetooth mesh for Android ADK
- Prerequisites for development
- Contents of the Bluetooth mesh Android delivery
- Code snippets and walkthrough

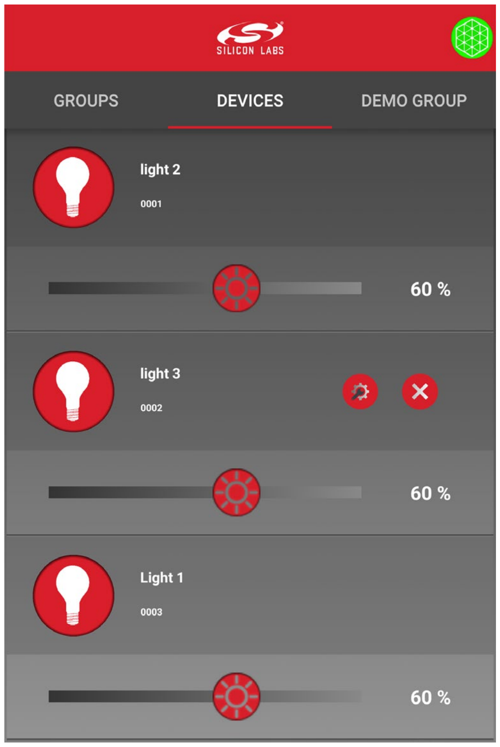


Table of Contents

1 Introduction

The Android Bluetooth LE stack does not have native support for Bluetooth mesh (API version 27 or older) and therefore Android devices cannot directly interact with Bluetooth mesh nodes using the Bluetooth mesh advertisement bearer. However, Bluetooth mesh specification 1.0 also defines a GATT bearer, which enables any Bluetooth LE capable devices to interact with Bluetooth mesh nodes over GATT. Android operating systems since API version 18 has included support for the Bluetooth GATT layer and hence it is possible to implement an Android application to provision, configure, and interact with Bluetooth mesh networks and nodes.

Silicon Labs provides a Bluetooth mesh stack not only for Blue Gecko Bluetooth SoCs and Modules but also for the Android operating system to enable development of Bluetooth mesh applications. The figure below shows a high level software block diagram illustrating what components and APIs a Bluetooth mesh-capable application requires and the source of the components (Android operating system or Silicon Labs).

The basic concept is that the Android Bluetooth stack and APIs are used to discover and connect Bluetooth LE devices and exchange data with them over GATT. The Silicon Labs Bluetooth mesh stack is used to manage the Bluetooth mesh-specific operations such as Bluetooth mesh security, device and node management, network, transport, and application layer operations.

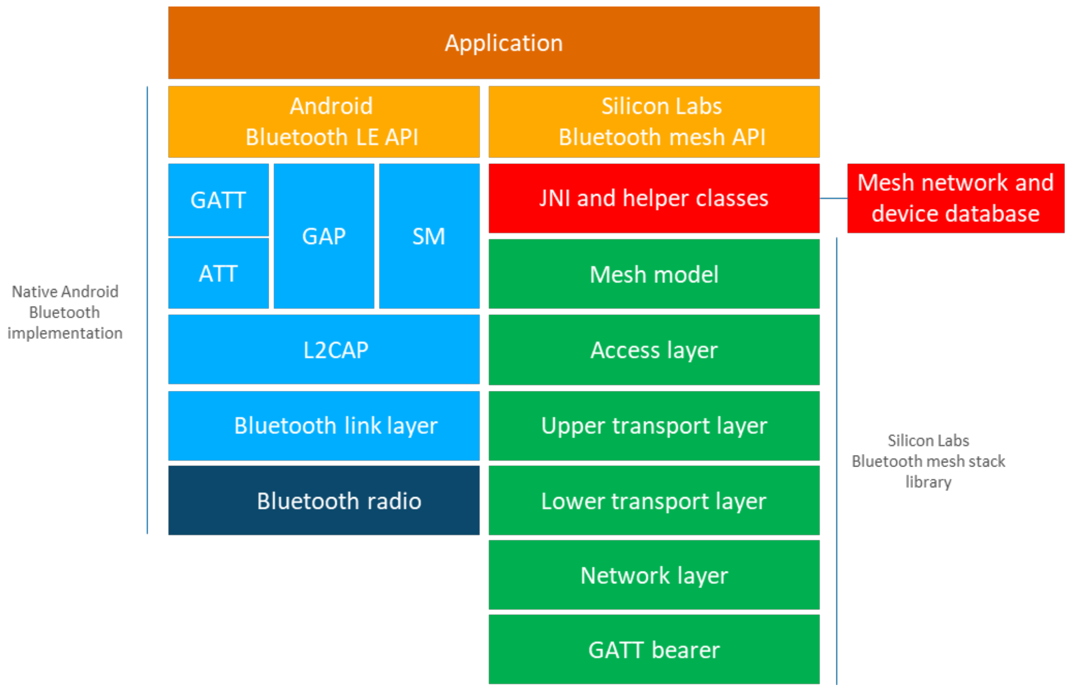


Figure 1-1: Bluetooth mesh-Capable Application Software Architecture Diagram

1.1 Features of the Bluetooth Mesh Android Stack

Table 1. Features of Silicon Labs Bluetooth mesh Stack for Android

Feature	Value
Bluetooth mesh stack version	1.2.0
Simultaneous GATT connections	1
Mesh bearers	GATT
Supported node types	Proxy Relay Low Power

Feature	Value
Maximum number of networks	1 (limitation of the Bluetooth mesh stack)
Maximum number of network keys	1 (limitation of the Android application)
Maximum number of application keys	1 (limitation of the Android application)
Maximum number of mesh nodes	255 (limitation of the Android application)
Supported mesh models	Configuration client Generic OnOff client, Generic Level client, Light Lightness client
Supported GATT services	Provisioning Proxy

1.2 Open Source Licenses Used

Feature	License	Comment
OpenSSL	See here	Used for AES and ECDH and other cryptographic algorithms. Provided as part of the Android ADK.
GSON	Apache License 2.0	Used to store and load the Bluetooth mesh and device database to the Android secure storage.

2 Prerequisites for Development

Before you can start developing Bluetooth mesh Android applications the following prerequisites must be met.

1. Basic understanding of Bluetooth LE and Bluetooth mesh 1.0 specifications and technology.
    1. Silicon Labs Bluetooth mesh learning center
    2. Silicon Labs Bluetooth mesh technology white paper
    3. Bluetooth mesh profile 1.0 specification
    4. Bluetooth mesh model 1.0 specification (application layer)
    5. Bluetooth 5.0 core specification

2. Simplicity Studio and Bluetooth mesh SDK 1.2.0 or newer are installed.
    1. Download
    2. Getting started

3. You have a Blue Gecko SoC (SLWSTK6020B) or module WSTKs to act as Bluetooth mesh nodes.
    1. Recommended SoCs and Modules are: EFR32BG13 or EFR32BG12 SoCs or BGM13P module because they have 512kB to 1024kB of flash available.
    2. SLWSTK6020B User Guide

4. Bluetooth mesh by Silicon Labs Android reference application is installed.
    1. Download

5. You have the Bluetooth mesh for Android ADK access and Java API documentation. Note that these are only available for licensed users. Contact Silicon Labs for access.

6. Android Studio is installed.
    1. Download
    2. Introduction

7. You have an Android phone or tablet preferably running Android 6.0 or newer.

8. You have an understanding of Bluetooth LE operation on Android as well as the JNI interface.
    1. Bluetooth LE documentation for Android
    2. Bluetooth LE API documentation
    3. Android JNI documentation

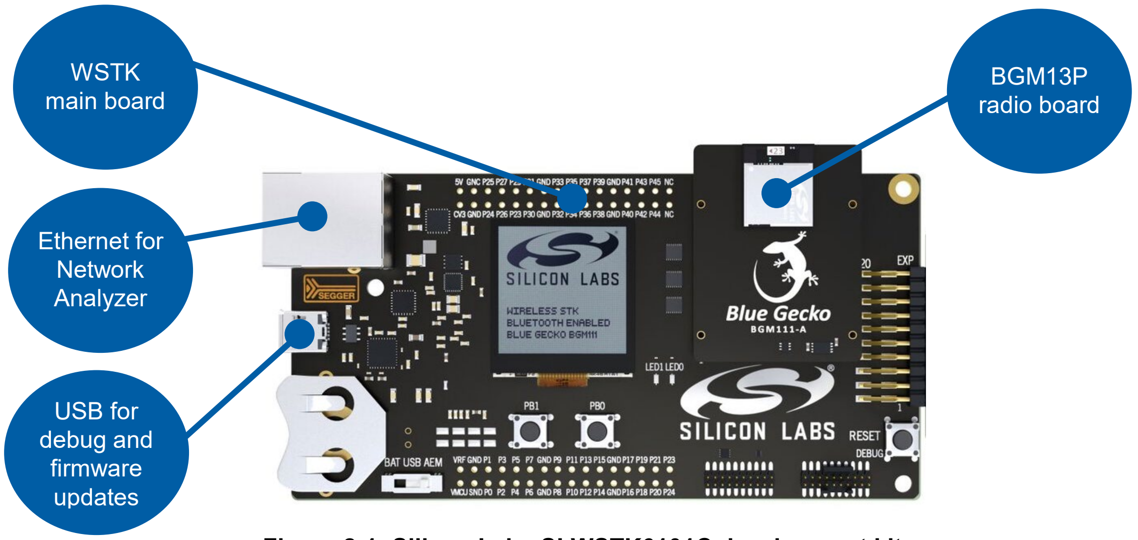

WSTK main board
BGM13P radio board
Ethernet for Network Analyzer
USB for debug and firmware updates

Figure 2-1: Silicon Labs SLWSTK6101C development kit

3 Contents of Bluetooth mesh for Android ADK

3.1 The Bluetooth mesh stack library

The Bluetooth mesh protocol stack is provided as a pre-compiled library and it implements the components show in Figure 2-1: Silicon Labs SLWSTK6101C development kit.

3.2 JNI Interface

The JNI interface provides the necessary abstraction between the Bluetooth mesh stack library and the application.

3.3 Bluetooth mesh Network and Device Database

The Bluetooth mesh device and network database contains all the information the provisioner stores about the devices and the network, including security keys, device addresses, supported elements, models and so on.

The device database is stored in the secure content of the application in AES encrypted GSON file format.

3.4 Bluetooth mesh Java API

Android applications are developed with Java programming language and the Silicon Labs Bluetooth mesh Java API provides an API for the application to interface with the Bluetooth mesh stack library.

The Java API provides access to the Bluetooth mesh stack library as well it contains necessary helper classes for Bluetooth mesh devices, networks, groups, models etc.

3.5 Reference Application Source Code

The Bluetooth mesh by Silicon Labs reference application is provided in source code as part of the delivery and it can be used as a reference implementation for application development.

3.6 Documentation

The delivery contains JavaDoc documentation for the Bluetooth mesh stack as well (this document.

4 Getting Started with Development

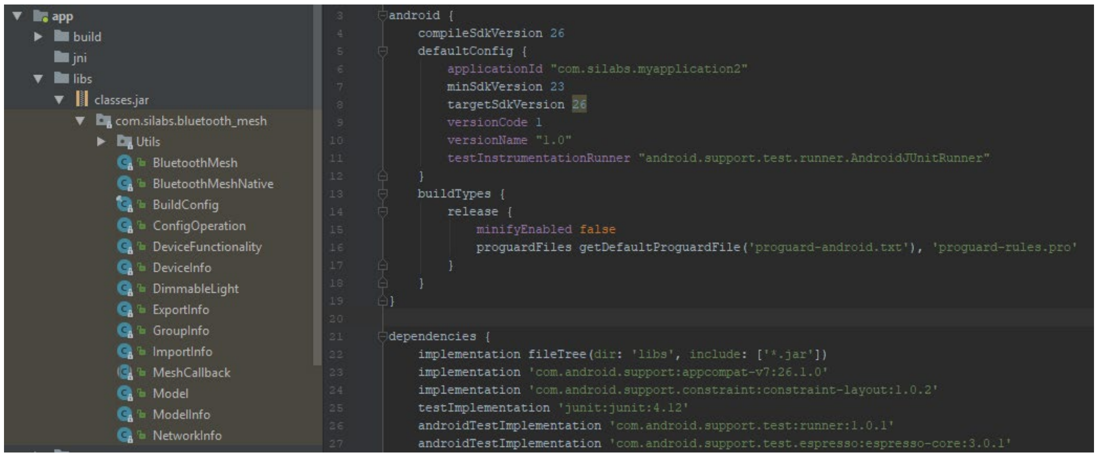
1. Install the Android Studio.

2. Unzip the Bluetooth mesh for Android ADK.
   1. Copy the .JAR file to your project folder (for example C:\Users\x\AndroidStudioProjects\BTMesh\app\libs).
   2. Copy the JNI libraries to your project folder (for example C:\Users\x\AndroidStudioProjects\BTMesh\app\src\main\jniLibs).

3. Open the Android studio.

4. Create a new project.
   1. Create the project for Phone and Tablet.
   2. Select at least API 23: Android 6.0 (Marshmellow).
   3. Create for example Empty Activity project.

5. Make sure the JAR file is imported to the project and the Bluetooth mesh classes are visible.

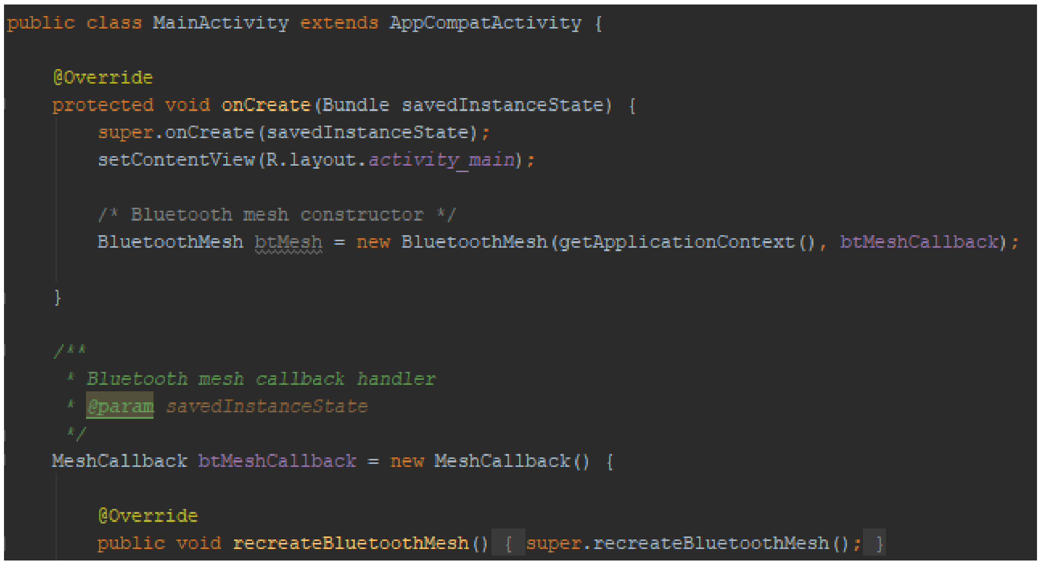


```
android {
    compileSdkVersion 26
    defaultConfig {
        applicationId "com.silabs.myapplication2"
        minSdkVersion 23
        targetSdkVersion 26
        versionCode 1
        versionName "1.0"
        testInstrumentationRunner "android.support.test.runner.AndroidJUnitRunner"
    }
    buildTypes {
        release {
            minifyEnabled false
            proguardFiles getDefaultProguardFile('proguard-android.txt'), 'proguard-rules.pro'
        }
    }
}

dependencies {
    implementation fileTree(dir: 'libs', include: ['*.jar'])
    implementation 'com.android.support:appcompat-v7:26.1.0'
    implementation 'com.android.support.constraint:constraint-layout:1.0.2'
    testImplementation 'junit:junit:4.12'
    androidTestImplementation 'com.android.support.test:runner:1.0.1'
    androidTestImplementation 'com.android.support.test.espresso:espresso-core:3.0.1'
```

Figure 4-1: Android Studio Project

6. Create a new **BluetoothMesh** object and a **MeshCallback** handler method as shown below.



```
public class MainActivity extends AppCompatActivity {

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);

        /* Bluetooth mesh constructor */
        BluetoothMesh btMesh = new BluetoothMesh(getApplicationContext(), btMeshCallback);

    }

    /**
     * Bluetooth mesh callback handler
     * @param savedInstanceState
     */
    MeshCallback btMeshCallback = new MeshCallback() {

        @Override
        public void recreateBluetoothMesh() { super.recreateBluetoothMesh(); }
```

Figure 4-2: BluetoothMesh Object and Callback Handler Method

7. Compile the project and make sure it compiles without errors.

4.1 File Location in the Bluetooth mesh SDK

```
$BT_MESH_SDK_ROOT/app/bluetooth/android/
```

4.2 Java JNI Libraries

The JNI libraries (JAVA Native Interface Libraries) are the interface between C/C++ libraries and the JVM and consequently the Android Project.

For the Bluetooth Mesh for Android ADK we use three libraries linked with each other, so the user application will use only one that depends on the other two. The user does not need to do anything with the library except to put them in the proper place in the Android Project file structure.

The standard place for the JNI libraries is:

```
$projectRoot/src/main/jniLibs
```

The user must copy the libraries to this path and they are then ready to be used.

If necessary, the user can change the path from the standard by adding the following lines in their gradle.build file:

```
android {
    sourceSets {
        main {
            jniLibs.srcDirs = ['<USER_PATH>']
        }
    }
}
```

4.3 JAR file

The JAR (class.jar) file contains all the .class files from the Bluetooth Mesh Android ADK.

The user must copy this file to:

```
$projectRoot/libs
```

and also add the following line of code to its build.gradle dependencies

```
implementation fileTree(include: "class.jar", dir: 'libs')
```

With these two steps the user now can create and use the classes from Bluetooth Mesh Android ADK in the Android Project.

5 Code Examples and Walkthrough

This section contains code examples and documentation about how to use the Bluetooth mesh Java API, Android Bluetooth API, and associated classes to perform various Bluetooth mesh operations like device provisioning, configuration, and others. The code examples are not meant to be complete working examples, but rather to provide the reader with an overview of the process.

5.1 BluetoothMesh

BluetoothMesh is the main object of the Bluetooth Mesh Android API. All the exchange of messages between the user application and the Bluetooth mesh library happens in this object (for example, Provisioning, Configuration, Controlling, Deletion, and so on).

The API is also provided with support objects that help the user manage the Bluetooth mesh network. These are:

- NetworkInfo
- GroupInfo
- DeviceInfo
- Model
- ConfigOperation

They will be explained later in this document.

To begin, initialize the BluetoothMesh object.

5.1.1 Initializing the BluetoothMesh Object

```
BluetoothMesh btMesh = new BluetoothMesh(Context context, MeshCallBack meshCallback);
```

The `context` parameter is the application context the user can access through `getApplicationContext()`.

NOTE: The stack cannot be initialized more than once. If it happens an exception is thrown. Make sure only one instance is running at a time. Do not initialize another instance before de-initializing the first one.

The **MeshCallback** object is the interface between the user application and the Bluetooth Mesh API.

5.1.2 MeshCallback

The **MeshCallback** object is the gate of communication from the Bluetooth mesh Android API and the user application. All operations that depend on the Bluetooth mesh network return their values to this callback. The callback must be implemented in the user application before initializing the BluetoothMesh object.

```
MeshCallback meshCallback = new MeshCallback() {
    @Override
    public void stateChanged(int state) {
        super.stateChanged(state);
        //User implementation here
    }

    @Override
    public void gattWrite(int gattHandle, byte[] send) {
        super.gattWrite(gattHandle, send);
        //User implementation here
    }
    @Override
    public void networkCreated(String name, int index, byte[] netKey) {
        super.networkCreated(name, index, netKey);
        //User implementation here
    }

    @Override
    public void appKeyCreated(String name, int netKey_index, int appKey_index , byte[] appKey) {
        super.appKeyCreated(name, netKey_index, appKey_index, appKey);
        //User implementation here
    }

    @Override
    public void didSuccessProvision(int meshAddress, byte[] deviceUuid, int status) {
        super.didSuccessProvision(meshAddress, deviceUuid, status);
        //User implementation here
    }

    @Override
    public void didReceiveDCD(byte[] dcd, int status) {
        super.didReceiveDCD(dcd, status);
        //User implementation here
    }

    @Override
    public void disconnectionRequest(int gattHandle) {
        super.disconnectionRequest(gattHandle);
        //User implementation here
    }

    @Override
    public void gattRequest(int gattHandle) {
        super.gattRequest(gattHandle);
        //User implementation here
    }

    @Override
    public void didExportRequest(Intent shareIntent) {
        super.didExportRequest(shareIntent);
        //User implementation here
    }

    @Override
    public void didOOBAuthdisplay(byte[]  uuid, int input_action, int input_size, byte[]  auth_data)
{
        super.didOOBAuthdisplay(uuid, input_action, input_size, auth_data);
        //User implementation here
    }

    @Override
    public void didOOBAuthRequest(byte[] uuid, int output_action, int output_size) {
        super.didOOBAuthRequest(uuid, output_action, output_size);
        //User implementation here
    }

    @Override
    public void didCompleteConfig(ConfigOperation previous_cfg, ConfigOperation next_cfg) {
        super.didCompleteConfig(previous_cfg, next_cfg);
        //User implementation here
    }
};
```

The user application must implement those methods as its own sake. Each of these items are discussed in more detail later in this document.

5.2 Supporting Objects

The supporting objects listed above are objects that allow the user to manage the Bluetooth mesh network in a simple way.

5.2.1 NetworkInfo

The **NetworkInfo** object contains the entire structure of the Bluetooth Mesh Network. It contains all the others support classes (except for ConfigOperation). The simple structure from the network is:

- NetworkInfo
    - Array of GroupInfo
    - Array of DeviceInfo

5.2.2 GroupInfo

GroupInfo object contains an Application Key and publish and subscribe addresses that will be used to configure the models inside a **DeviceInfo** object. This object also contains an Array of **DeviceInfo** objects that will reply to this address when a group message is sent. The simple structure is:

- GroupInfo
    - Array of **DeviceInfo** (gets the objects from the network)

5.2.3 DeviceInfo

DeviceInfo object contains all the information about a Bluetooth mesh device in a network. It is created immediately after provision a device and it is actually defined after receiving the device composition data (DCD), which tells the features available in the node and also the models available for configuration (see section Request DCD (Device Composition Data).

After the model is configured, it will be added to a new list of elements inside the **Config** object.

- **DeviceInfo**
    - DCD
        - Features (available)
        - Elements (available)
        - Models (available)
    - Config
        - NetKeys (configured networks) //limited to 1 in this version
        - AppKeys (configured groups) //limited to 1 in this version
        - Elements (constructor generates elements as in the DCD, but without Models)
        - Models (configured models)

5.2.3.1 Model

Model is an object created when a **DeviceInfo** object receives the device composition data (DCD). There is not much interaction with this object in this version, but the user must use the object as a parameter for configuration.

5.2.3.2 ConfigOperation

ConfigOperation is an object created by the BluetoothMesh class every time the user sends one of the configuration commands. This object contains all the information about the configuration process, like **NetworkInfo**, **GroupInfo**, **DeviceInfo**, **Model**, **Element ID**, **kind** (what was the command, for example `addToGroup`), and **status**.

5.3 Provisioning

5.3.1 Setting up BLE GATT Connections

5.3.1.1 Scanning Bluetooth Mesh Devices

To filter for Bluetooth Mesh devices, the user must add a filter to the **BluetoothLeScanner** object before starting the scanner.

```
ScanSettings settings = new ScanSettings.Builder()
                        .setScanMode(ScanSettings.SCAN_MODE_BALANCED)
                        .build();
List<ScanFilter> filters = new ArrayList<>();
```

```
ParcelUuid meshServ = ParcelUuid.fromString(BluetoothMesh.meshUnprovisionedService.toString());
ScanFilter filter = new ScanFilter.Builder().setServiceUuid(meshServ).build();
filters.add(filter);
bluetoothLeScanner.startScan(filters , settings, scanCallback)
```

5.3.1.2 Setting Connection Parameters

It is highly recommended to set the connection parameters of the GATT connection:

```
gatt.requestConnectionPriority(BluetoothGatt.CONNECTION_PRIORITY_HIGH); //No Callback
gatt.requestMTU(69); //Get confirmation in "onMtuChanged" (Range 23 - 69(optimum))
```

5.3.1.3 Discovering Services

Services can be discovered on the MTU changed callback. It is tup to the user when to call this method:

```
gatt.discoverServices(); //Get confirmation in "onServicesDiscovered"
```

5.3.1.4 Enabling Notification to the Characteristic

After discovering services the user can set notification to the Mesh Out Characteristic:

```
public void enableNotification()
   {
       BluetoothGatt gatt = bluetoothGatt;
       if(gatt == null)  //Check gatt connection
       {
           return;
       }
       BluetoothGattService btServ = gatt.getService(BluetoothMesh.meshUnprovisionedService);
       if (btServ == null) //Check mesh service
       {
           return;
       }

       BluetoothGattCharacteristic                             btChar                            =
btServ.getCharacteristic(BluetoothMesh.meshUnprovisionedOutChar);
       if(btChar == null) //Check mesh char
       {
           return;
       }
       BluetoothGattDescriptor                              descriptor                           =
btChar.getDescriptor(BluetoothMesh.meshOutCharDescriptor);
       if(descriptor == null) //Check mesh descriptor
       {
           return;
       }
       gatt.setCharacteristicNotification(btChar,true);
       descriptor.setValue(BluetoothGattDescriptor.ENABLE_NOTIFICATION_VALUE);
       gatt.writeDescriptor(descriptor);
   }
```

It is also important to set the method for writing messages to the un-provisioned device:

```
public void writeGattChar(int gattHandle, byte[] writeArray)
{
    if(mGatt == null) {
        Log.e("writeGattChar", "Device is not connected");
        return;
    }
    BluetoothGattService meshService = mGatt.getService(BluetoothMesh.meshUnprovisionedService);
    if(meshService == null) {
       Log.e("writeGattChar", "Device does not contain meshService");
```

```
        return;
    }
    BluetoothGattCharacteristic meshWrite =
meshService.getCharacteristic(BluetoothMesh.meshUnprovisionedInChar);
    if(meshWrite == null) {
        Log.e("writeGattChar", "Device does not contain meshInCharacteristic");
        return;
    }
    meshWrite.setValue(writeArray);
    meshWrite.setWriteType(BluetoothGattCharacteristic.WRITE_TYPE_NO_RESPONSE);
    mGatt.writeCharacteristic(meshWrite);
}
```

5.3.1.5 The GATT Handle

The integer `gattHandle` is for future implementation. For now the ADK does not support multiple GATT connections. The user should use "0" whenever it is requested as a parameter.

With all the above information, the Bluetooth GATT connection is ready for provisioning purposes.

5.3.1.6 Initialization Callback

After creating the btMesh object, the user will receive a confirmation that the stack was initialized properly in stateChanged in **Mesh-Callback**.

```
@Override
public void stateChanged(int state) {
    super.stateChanged(state);
    switch(state)
    {
        case BluetoothMesh.INITIALISED:
        //User implementation here
        break;
        case BluetoothMesh.DEINITIALISED:
        //User implementation here
        break;
    }
}
```

To provision a device, the user must create a network. Information about the network will be passed as a parameter of provisionDevice.

5.3.2 Creating a Bluetooth Mesh Network

Create a network using:

```
NetworkInfo netInfo =  btMesh.createNetwork(String name, byte[] net_key);
```

`byte[] app_key` is a byte array with size = 16 defined by the user application.

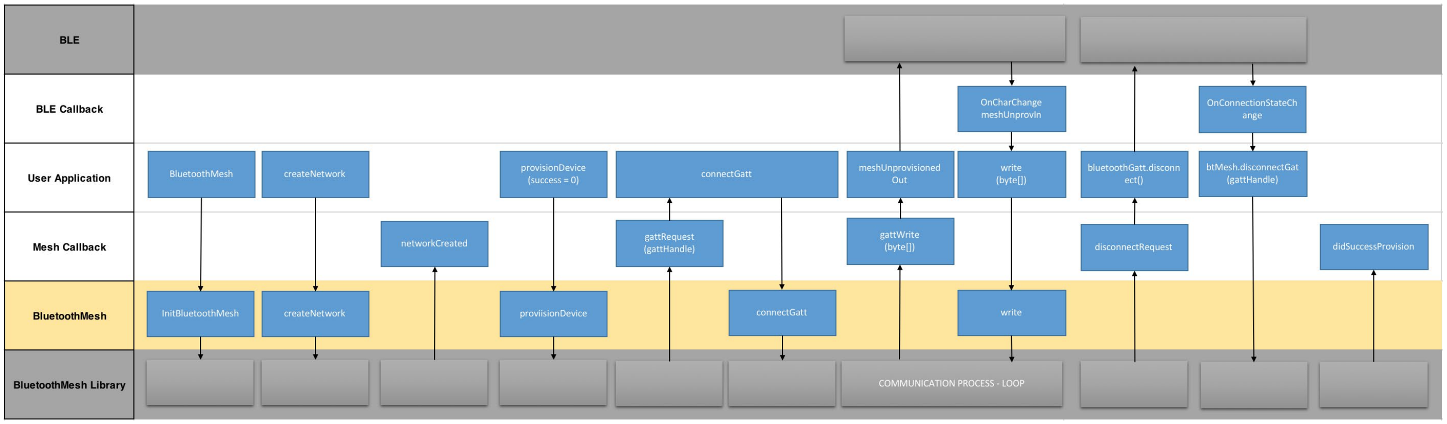
The application will receive and

```
@Override
     public void networkCreated(String name, int index, byte[] netKey) {
         super.networkCreated(name, index, netKey);
         NetworkInfo defaultNetwork = new NetworkInfo();
         defaultNetwork.setName(name);
         defaultNetwork.setNetwork_key(netKey);
         defaultNetwork.setNetID(index);
         bluetoothMesh.saveNetworkDB(defaultNetwork);
         createDemoGroup(defaultNetwork);
  }
```

5.3.3 Provisioning a Mesh Device to a Bluetooth mesh Network

To provision a device use:

```
btMesh.provisionDevice(NetworkInfo netInfo, byte[] deviceUuid, int gattHandle, int mtu);
```



```
@Override
public void disconnectionRequest(int gattHandle) {
    super.disconnectionRequest(gattHandle);
    BluetoothGatt.disconnect();
    btMesh.disconnectGatt(gattHandle);
}
```

After a successful provisioning, the GATT connection must be dropped. After disconnection the node will start to advertise as a proxy node for 60 seconds.

```
@Override
public void didSuccessProvision(int meshAddress, byte[] deviceUuid, int status) {
     super.didSuccessProvision(meshAddress, deviceUuid, status);
     //User implementation here
}
```

The provisioning session has ended when the user application receives the didSucessProvision callback If the status equals 0, the provisioning session was successful. With the `meshAddress` and `deviceUuid` the user is now able to create a **DeviceInfo** object, and save it to the **NetworkInfo** object.

5.3.4 Add Device to Network

Create a **DeviceInfo** object:

```
DeviceInfo exampleDevice = new DeviceInfo(String name, byte[] deviceUuid);
```

(Name is optional → The library does care about name, it is used as an application and UI identifier)

Add the **DeviceInfo** object to the **NetworkInfo** object and save it into the database:

```
exampleDevice.addToNetwork(NetworkInfo netInfo, int meshAddress);
netInfo.devicesInfo.add(exampleDevice);
btMesh.saveNetworkDB(netInfo);
```

6 Configuring a Mesh Node

6.1 Connecting with Proxy

Start the BluetoothLeScanner:

```
BluetoothLeScanner bluetoothLeScanner = bluetoothAdapter.getBluetoothLeScanner();
bluetoothLeScanner.startScan(scanCallback);
```

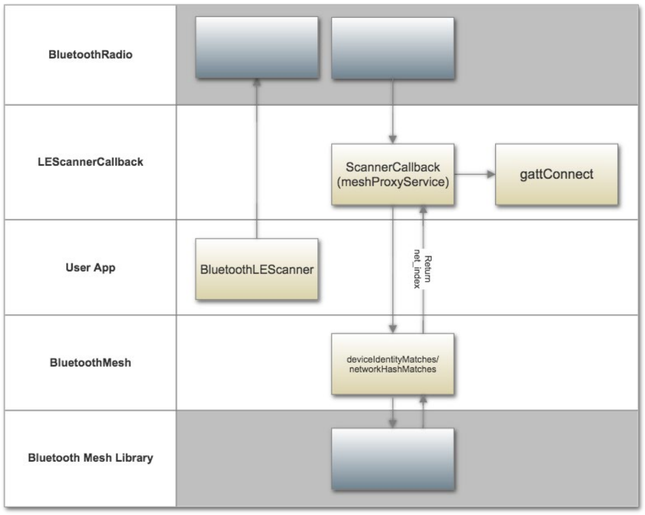
Set the scanCallback:

```
private ScanCallback scanCallback = new ScanCallback() {
       @Override
       public void onScanResult(int callbackType, ScanResult result) {
           // First go through devices which don't have DCD entries yet, they might be
           // advertising with Identity instead of Network beacons, for example right
           // after provisioning
           boolean idMatch = false;
           boolean netMatch = false;
           ParcelUuid meshProxyServ = Par-
celUuid.fromString(BluetoothMesh.meshProxyService.toString());
           if (result.getScanRecord()!= null
                   && result.getScanRecord().getServiceUuids()!= null
                   && result.getScanRecord().getServiceUuids().contains(meshProxyServ)) {

               for (DeviceInfo devInfo : netInfo.devicesInfo) {
                   if (d.dcd() == null) {
                       if (btMesh.deviceIdentityMatches(result.getScanRecord().getBytes(),  devInfo)
>= 0)
                       {
                           identityMatchResults.add(result);
                           idMatch = true;
                           break;
                       }
                   }
               }
               // If we already have an Identity match, it won't match the Network beacon
               if (!idMatch && bluetoothMesh.networkHashMatches(netInfo, re-
sult.getScanRecord().getBytes()) >= 0)
               {
                   networkMatchResults.add(result);
                   netMatch = true;
               }

               if (!idMatch && !netMatch) return;

              //STOP SCAN AND CONNECT GATT WITH DEVICE
           }
       }
   };
```



If a device was just provisioned, it has a 60-second window of GATT advertising waiting to be configured. To find this device:

```
btMesh.deviceIdentityMatches(byte [] adv_data, devInfo);
```

To only connect with the closest proxy:

```
btMesh.networkHashMatches(netInfo, byte[] adv_data);
```

6.1.1 BLE GATT Connections for proxy

It iss highly recommended to set the connection parameters of the GATT connection:

```
gatt.requestConnectionPriority(BluetoothGatt.CONNECTION_PRIORITY_BALANCED); //No Callback
gatt.requestMTU(69); //Get confirmation in "onMtuChanged" (Range 23 - 69(optimum))
```

Services can be discovered on the MTU changed callback; it is up to the user when to discover them.

```
gatt.discoverServices(); //Get confirmation in "onServicesDiscovered"
```

After discovering services, the user can set notification to the outchar.

```
void setNotification(BluetoothGatt gatt)
{
   if(gatt == null)
   {
       return;
   }
   BluetoothGattService btServ = gatt.getService(BluetoothMesh.meshProxyService);
   if(btServ == null)
   {
       return;
   }
   BluetoothGattCharacteristic btChar = btServ.getCharacteristic(BluetoothMesh.meshProxyOutChar);
   if(btChar == null)
   {
       return;
   }
   BluetoothGattDescriptor descriptor = btChar.getDescriptor(BluetoothMesh.meshOutCharDescriptor);
   if(descriptor == null)
   {
       return;
   }
   gatt.setCharacteristicNotification(btChar, true);
   descriptor.setValue(BluetoothGattDescriptor.ENABLE_NOTIFICATION_VALUE);
   gatt.writeDescriptor(descriptor);
}
```

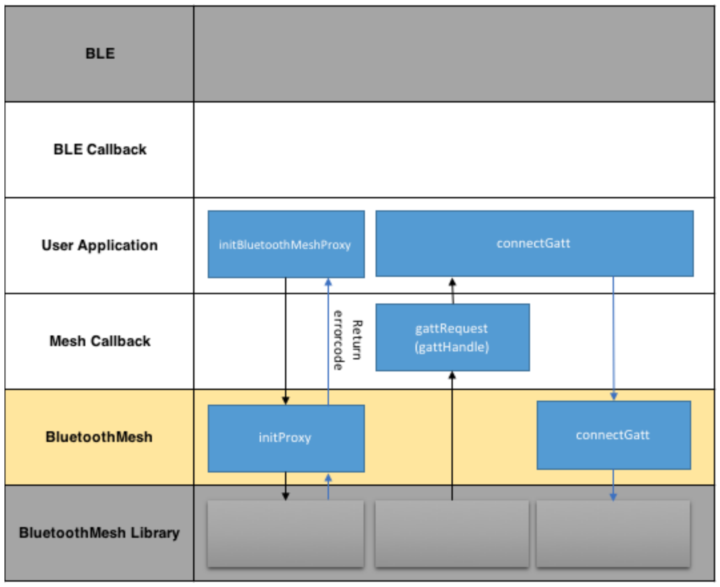
The user must also set the method to write to the proxy characteristic:

```
public void writeGattChar(int gattHandle, byte[] writeArray)
{
    if(mGatt == null) {
        Log.e("writeGattChar", "Device is not connected");
        return;
    }
    BluetoothGattService meshService = mGatt.getService(BluetoothMesh.meshProxyService);
    if(meshService == null) {
       Log.e("writeGattChar", "Device does not contain meshService");
       return;
    }
    BluetoothGattCharacteristic                             meshWrite                              =
meshService.getCharacteristic(BluetoothMesh.meshProxyInChar);
    if(meshWrite == null) {
        Log.e("writeGattChar", "Device does not contain meshInCharacteristic");
        return;
    }
    meshWrite.setValue(writeArray);
    meshWrite.setWriteType(BluetoothGattCharacteristic.WRITE_TYPE_NO_RESPONSE);
    mGatt.writeCharacteristic(meshWrite);
}
```

Now the Bluetooth GATT connection is ready to be used for proxy purposes.

6.1.2 Initializing Proxy

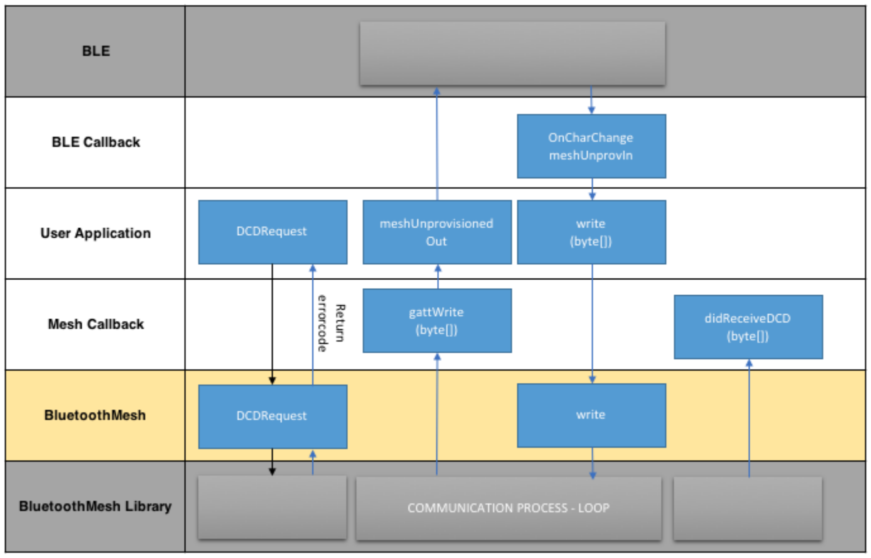
```
btMesh.initBluetoothMeshProxy(int gattHandle, int mtu);
```



6.1.3 Request DCD (Device Composition Data)

Once the user application has a GATT connection with the node and proxy is initialized, the application can request DCD. This process should be done only once with each device.

```
btMesh.DCDRequest(DeviceInfo devInfo);
```



The user application must save the `byte[]` received in `didReceiveDCD` into the respective **DeviceInfo** object:

```
exampleDevice.setDcd(byte[] dcd);
btMesh.saveNetworkDB(netInfo);
```

After receiving the DCD, the **DeviceInfo** object contains a DCD object. This object contains the information of what features (proxy, relay, friend, low power) and the elements the device supports. The elements contain the Models that will be used to configure a device.

DCD

- features
- elements
    - models

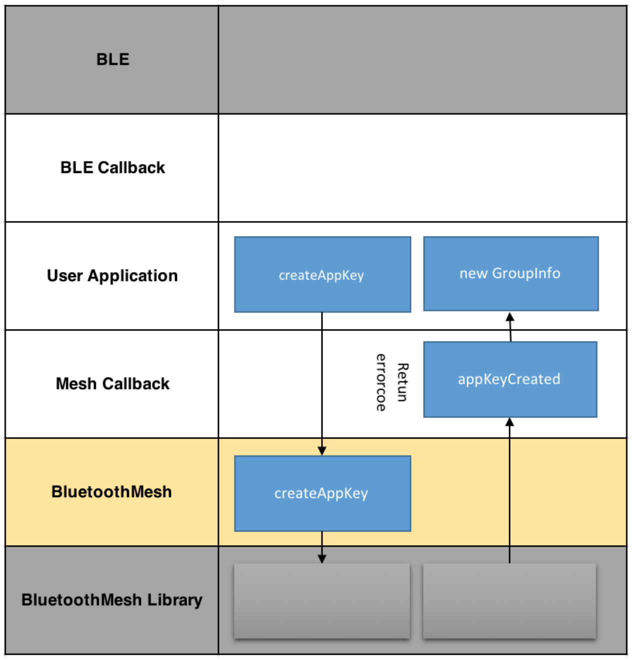
6.1.4 Create a Group

NOTE: A Group is a concept created for this ADK. It is an appKey holder that contains its own publish and subscribe addresses. Therefore configuring a node's model to a group is the same as giving the model the group's publish and subscribe addresses.

To configure a device, the user application must have a group (appKey). To create a group:

```
btMesh.createAppkey(String name, NetworkInfo net, byte[] app_key);
```

`byte[] app_key` is a byte array with size = 16 defined by the user application.



The application receives the appKeyCreated in the **MeshCallback** object:

```
@Override
       public void appKeyCreated(String name, int netKey_index, int appKey_index, byte[] appKey){
           super.appKeyCreated(name, netKey_index, appKey_index, appKey);
           NetworkInfo netInfo = btMesh.getNetworkbyID(netKey_index);
           int pub_address = 0xc000 + appKey_index*2;
           int sub_address = 0xc000 + appKey_index*2 + 1;
           GroupInfo new_group = new GroupInfo(name,
                   netInfo,
                   appKey_index, appKey,
                   pub_address,
                   sub_address);
           netInfo.addGroup(new_group);
           btMesh.bindLocalModels(netInfo, new_group);
           btMesh.saveNetworkDB(netInfo);

       }
```

The following explains the code above.

The GroupInfo constructor receives the following parameters.

```
GroupInfo exampleGroup = new Group(String name,
                                   NetworkInfo net,
                                   app_key_index,
                                   app_key,
                                   pub_address,
                                   sub_address);
```

Publish and Subcribe are derivative from the the appKeyIndex:

```
int pub_address = 0x0c00 + app_key_index*2;
int sub_address = 0x0c00 + app_key_index*2+1;
```

The group is saved to the network:

```
netInfo.addGroup(exampleGroup);
btMesh.bindLocalModels(netInfo, exampleGroup);
btMesh.saveNetworkDB(netInfo);
```

6.1.5 Configuration Commands

The API is limited in a few types of configurations based on the following configuration commands:

- `EnableProxy (cfg_state_enable_proxy)`
- `DisableProxy (cfg_state_disable_proxy)`
- `EnableRelay (cfg_state_enable_relay)`
- `DisableRelay (cfg_state_disable_relay)`
- `EnableFriend (cfg_state_enable_relay)`
- `DisableFriend (cfg_state_disable_relay)`
- `AddToGroup (cfg_state_add_key, cfg_state_bind, cfg_state_pub, cfg_state_sub)`
- `RemoveFromGroup(cfg_state_clear_sub, cfg_state_remove_old_key)`
- `FactoryReset (cfg_state_reset_device)`

All these methods returns callbacks to the same *didCompleteConfiguration* callback. This gives the user application two **ConfigOperation** objects, previous and next operations, which respectively tells the user the status of the previous configuration operation and provides information about the next one.

```
didCompleteConfig(ConfigOperation previous_config, ConfigOperation next_config);
```

6.1.6 Understanding ConfigOperation

The user can send configuration requests in a row. The API fills up a queue of config operations, and executes them one at a time when no configuration is being applied.

The **ConfigOperation** object is created by the *didCompleteConfig* callback. The user can get the following values from the object:

```
@Override
 public void didCompleteConfig(ConfigOperation previous_cfg, ConfigOperation next_cfg) {
    super.didCompleteConfig(previous_cfg, next_cfg);
    //Example of usage of ConfigOperation
    DeviceInfo devInfo = previous_cfg.device();
    GroupInfo groupInfo = previous_cfg.group();
    NetworkInfo netInfo = previous_cfg.network();
    Model model = previous_cfg.model();
    int kind = previous_cfg.kind();
    int status = previous_cfg.status();
}
```

6.1.6.1 Status

The Integer status is applied only for the **previous_cfg** object, and tells the user the status of the operation.

Inside the ConfigOperation class there is a Status `enum` that classifies the status numbers by name. To access the names, create a switch like:

```
switch (cfg.convertStatus(status))
{
    case mesh_foundation_status_success:
        //Operation Successful
        //User implementation
        break;
    case mesh_foundation_status_invalid_publish_params:
        //Device doesn't support publish
        //User implementation
        break;
    case mesh_foundation_status_timeout:
        //Timeout operation
        //User implementation
        break;
    .
    .
    .
    default:
        break;
}
```

6.1.6.2 Next Configuration

In the *didCompleteConfig* callback, the user will receive the next **ConfigOperation**, which can be used to determine the next operation, and apply it with `applyNextConfig()`:

```
@Override
public void didCompleteConfig(ConfigOperation previous_cfg, ConfigOperation next_cfg) {
    super.didCompleteConfig(previous_cfg, next_cfg);
    //Trigger the next configuration.
    btMesh.applyNextConfig();
    //The API will not do anything if the previous operation is in progress.
}
```

If the user does not want to apply any of the next configuration (for example if they received an error from the **previous_cfg** object), it is possible to cancel the pending configurations with:

```
@Override
public void didCompleteConfig(ConfigOperation previous_cfg, ConfigOperation next_cfg) {
    super.didCompleteConfig(previous_cfg, next_cfg);
    //This method cancel all pending operations
    btMesh.cancelPendingCfgs();
}
```

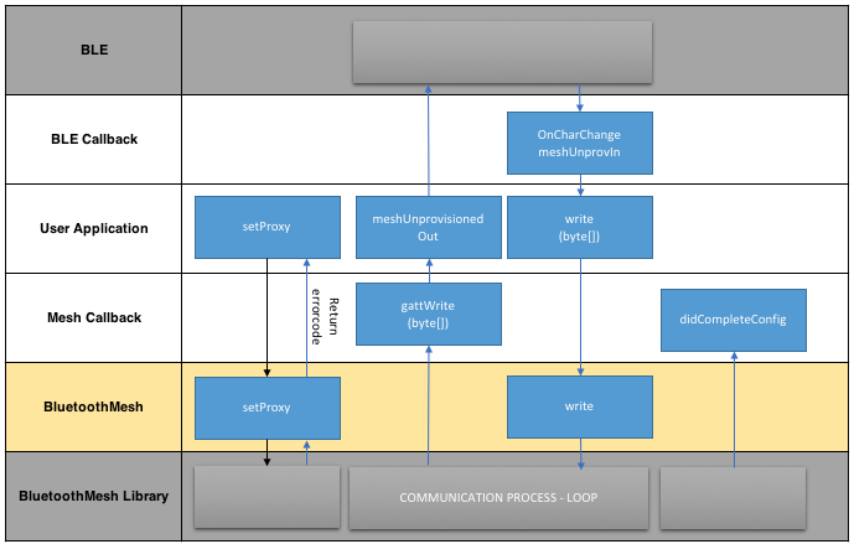
6.1.7 Set Proxy

Assuming the user is connected with a proxy device, and already has the device composition data, the user can check for the proxySupport feature from:

```
devInfo.DCD().proxySupport();
```

If the feature is available, the user can enable/disable it through:

```
btMesh.setProxy(devInfo, netInfo, boolean enable/disable);
```



Receive a callback from the operation in:

```
didCompleteConfig(ConfigOperation previous_config, ConfigOperation next_config);
```

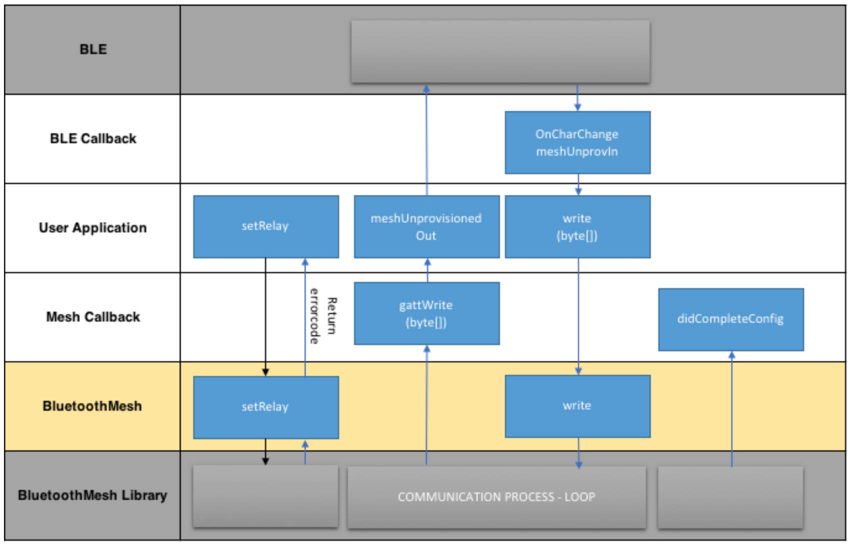
6.1.8 Set Relay

Assuming the user is connected with a proxy device, and already has the device composition data, the user can check for the relaySupport feature from:

```
devInfo.DCD().relaySupport();
```

If the feature is available, the user can enable/disable it through:

```
btMesh.setRelay(devInfo, netInfo , boolean enable/disable);
```



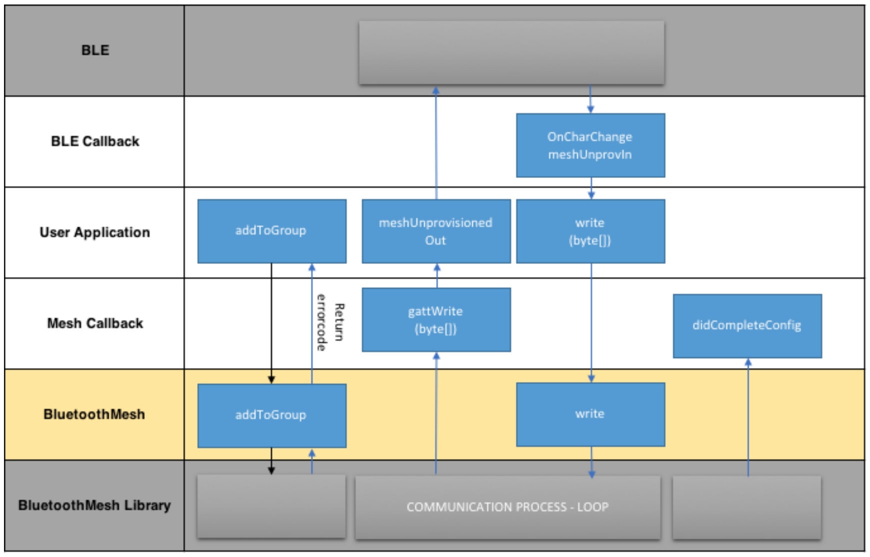
Receive a callback from the operation in:

```
didCompleteConfig(ConfigOperation previous_config, ConfigOperation next_config);
```

6.1.9 Configuring a mesh node

To configure a node, call:

```
btMesh.addToGroup(DeviceInfo devInfo,
                  int element,
                  NetworkInfo netInfo,
                  GroupInfo groupInfo,
                  Model model);
```



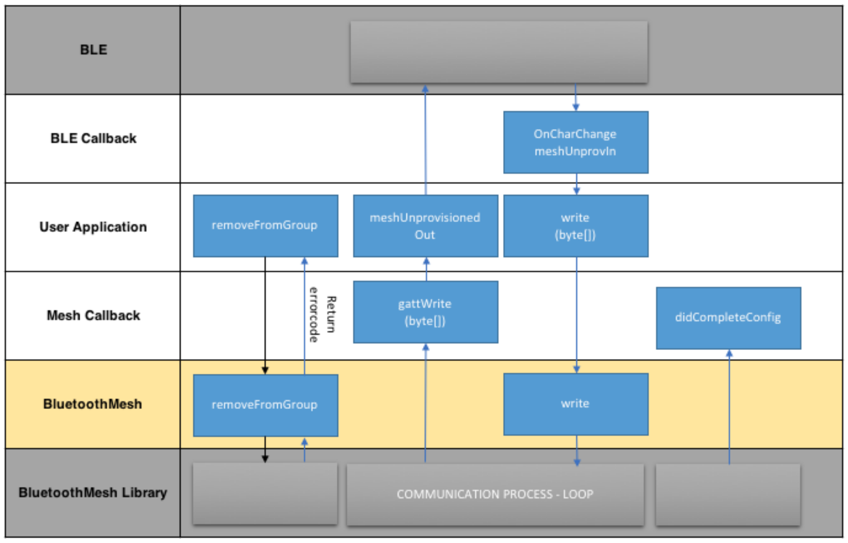
Receive a callback from the operation in:

```
didCompleteConfig(ConfigOperation previous_config, ConfigOperation next_config);
```

6.1.10 Removing a configuration from the mesh node

To remove a configuration from a node, call:

```
btMesh.removeFromGroup(DeviceInfo devInfo,
                       int element,
                       NetworkInfo netInfo,
                       GroupInfo groupInfo,
                       Model model);
```



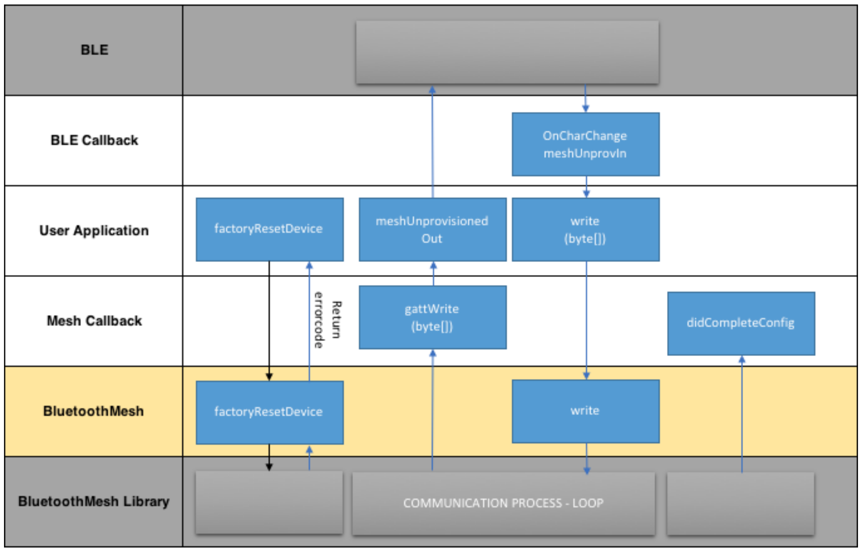
Receive a callback from the operation in:

```
didCompleteConfig(ConfigOperation previous_config, ConfigOperation next_config);
```

6.1.11 Factory Resetting a Bluetooth Mesh Node

To factory reset the device, removing it from the database (non-key refresh):

```
btMesh.factoryResetDevice(DeviceInfo devInfo, NetworkInfo netInfo);
```



Receive a callback from the operation in:

```
didCompleteConfig(ConfigOperation previous_config, ConfigOperation next_config);
```

6.2 Controlling a Mesh Node

As of this writing, the API supports three server models called controllers in this document.

6.2.1 Generic On Off Server

If the user has the "Generic On Off Server" model configured (through addToGroup),the device can be controlled using the following method:

```
btMesh.onOffSet(DeviceInfo devInfo,
                GroupInfo groupInfo,
                boolean status,
                int transition_ms,
                int delay_ms,
                boolean request_reply,
                boolean is_final);
```

In a future implementation, if the boolean `request_reply` is True the user will receive a callback from the library with the changed status.

6.2.2 Generic Level Server

If the user has the "Generic Level Server" model configured (through addToGroup), the device can be controlled using the following method:

```
btMesh.levelSet(DeviceInfo devInfo,
                GroupInfo groupInfo,
                int status,
                int transition_ms,
                int delay_ms,
                boolean request_reply,
                boolean is_final);
```

In a future implementation, if the boolean `request_reply` is True the user will receive a callback from the library with the changed status.

6.2.3 Lightning Lightness Server

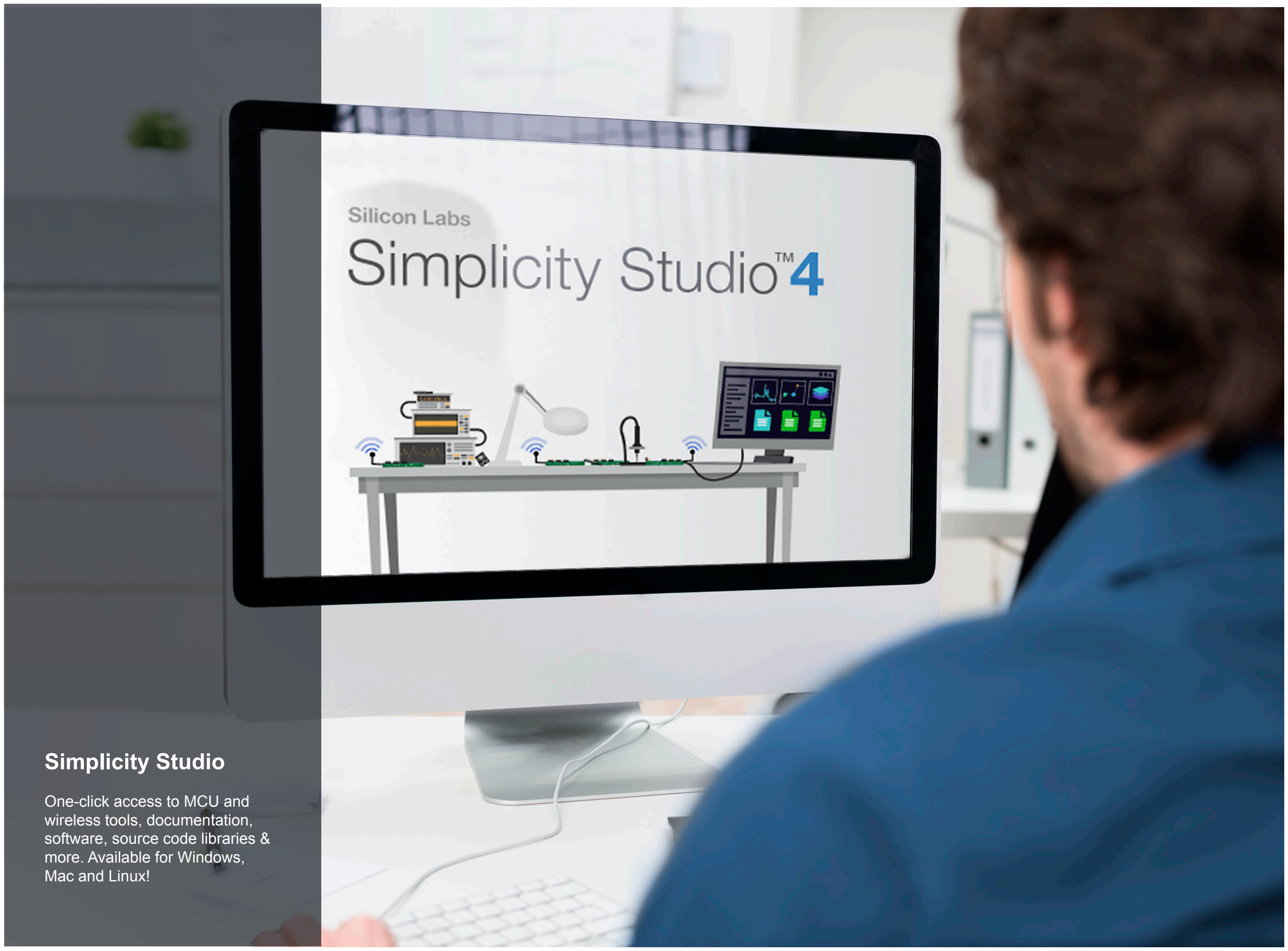
If the user has the "Lightning Lightness Server" model configured (through addToGroup), the device can be controlled using the following method:

```
btMesh.dimmableSet(DeviceInfo devInfo,
                   GroupInfo groupInfo,
                   int status,
                   int transition_ms,
                   int delay_ms,
                   boolean request_reply,
                   boolean is_final);
```

In a future implementation, if the boolean `request_reply` is True the user will receive a callback from the library with the changed status.

7 Support

Development support is available through the Silicon Labs online support.



Simplicity Studio

One-click access to MCU and wireless tools, documentation, software, source code libraries & more. Available for Windows, Mac and Linux!

IoT Portfolio
www.silabs.com/IoT

SW/HW
www.silabs.com/simplicity

Quality
www.silabs.com/quality

Support and Community
community.silabs.com

SILICON LABS

Silicon Laboratories Inc.
400 West Cesar Chavez
Austin, TX 78701
USA

http://www.silabs.com